Iterative Computation

Computational Methods, Oct. 2017, Kenji Doya

Many algorithm involve iterative computation until a solution is found with a desired accuracy.

Also many mathematical models are formulated as a mapping from the current state to the next state, which gives discrete-time dynamics.

References:

- Python Tutorial chapter 4: Control Flow Tools
- Wikipedia: Newton's method, Logistic map, Iterated function system

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Newton's method

For a linear equation $Ax = b$, or $Ax - b = 0$, the solution can be given by the inverser matrix as $x = A^{-1}b$.
For a general nonlinear equation $f(x) = 0$, the solution involves iterative computation, most typically by Newton's method:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

starting from an initial guess x_0.

Example

Define a plynomial function and its derivative.

In [2]:
```
def poly(x, a, deriv=True):
    """Polynomial a[0] + a[1]*x + ... + a[n]*x**n
        and its derivative"""
    y = 0  # output
    dy = 0    # derivative
    for i, ai in enumerate(a):  # gives the index and item
        y = y + ai*x**i
        if deriv and i>0:
            dy = dy + ai*i*x**(i-1)
    if deriv:
        return(y, dy)
    else:
        return(y)
```

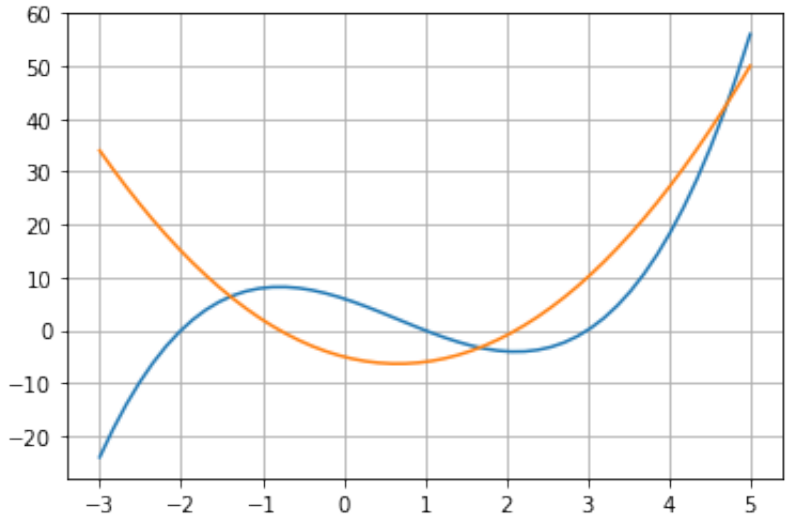
In [3]:
```
poly(3, [3, 2, -1])
```

Out[3]: (0, -4)

In [4]:
```
# f(x) = (x-3)(x-1)(x+2) = 6 - 5x -2x^2 + x^3
a = np.array([6, -5, -2, 1])
#a = np.random.randn(4) # random coefficients
print(a)
x = np.linspace(-3, 5, 50)
y, dy = poly(x, a)
plt.plot(x, y, x, dy)
plt.grid('on')
```

```
[ 6 -5 -2  1]
```

```
/Users/doya/anaconda/lib/python3.6/site-packages/matplotlib/cbook/__
init__.py:424: MatplotlibDeprecationWarning:
Passing one of 'on', 'true', 'off', 'false' as a boolean is deprecat
ed; use an actual boolean (True/False) instead.
  warn_deprecated("2.2", "Passing one of 'on', 'true', 'off', 'false
' as a "
```

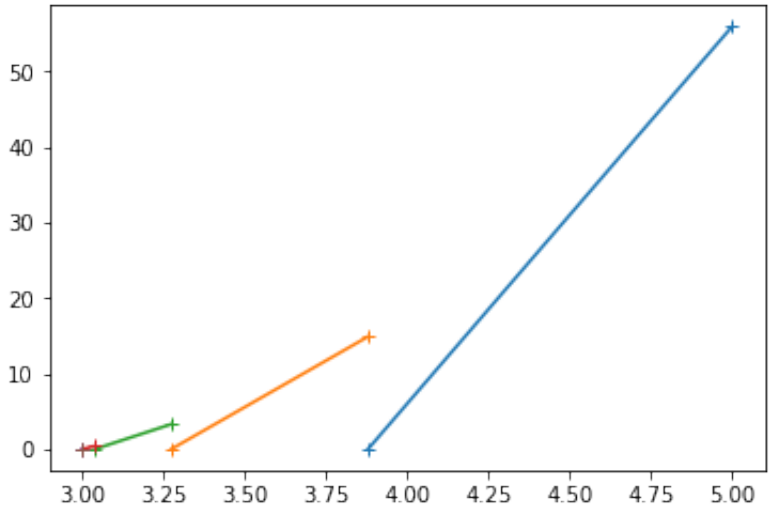


In [5]:
```
def newton(f, x0, a, target=1e-6):
    """Newton's method.
        f: should also return df/dx
        x: initial guess
        a: parameter for f(x,a)
        target: accuracy target"""
    y, dy = f(x0, a)
    while abs(y) > target:
        x = x0 - y/dy    # new x
        plt.plot([x0,x], [y,0], '-+')
        y, dy = f(x, a)  # new y
        print(x, y)
        x0 = x
    return(x, y)
```

In [6]:
```
newton(poly, 5, a)
```

```
3.88 14.902272000000004
3.275278535255161 3.3040933880698837
3.04063338160955 0.4179584066291824
3.0011057194602886 0.011065754263430705
3.0000008547777255 8.547782368140133e-06
3.0000000000005116 5.115907697472721e-12
```

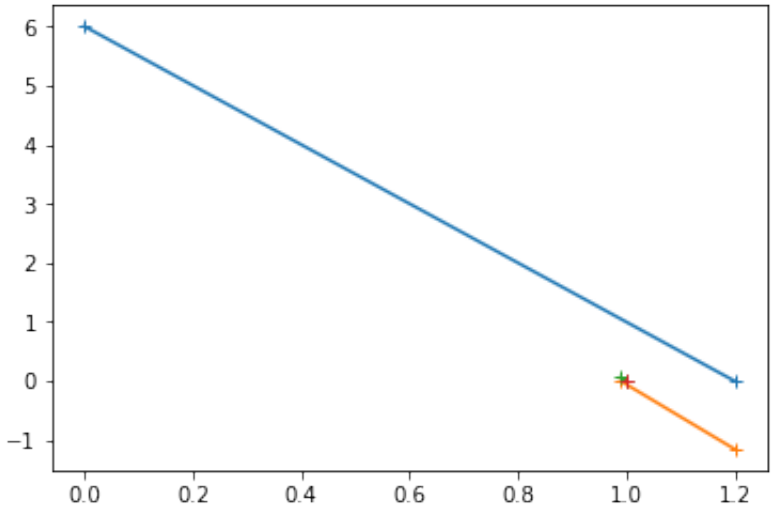
Out[6]: (3.0000000000005116, 5.115907697472721e-12)



In [7]:
```
1 newton(poly, 0, a)
```

```
1.2 -1.1520000000000001
0.9897810218978101 0.06141722898411894
0.9999830081212514 0.00010195156121095561
0.9999999999518813 2.8871205426383995e-10
```

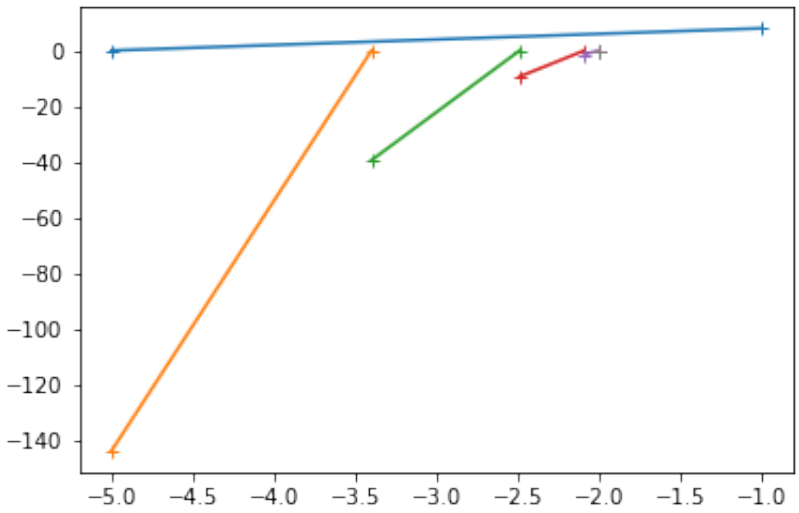
Out[7]: (0.9999999999518813, 2.8871205426383995e-10)



In [8]:
```
1 newton(poly, -1, a, 1e-12)
```

```
-5.0 -144.0
-3.4 -39.42399999999999
-2.48909426987061 -9.367117494376247
-2.0912240478732786 -1.435694484118736
-2.004130711225782 -0.06209724106997072
-2.0000090695520782 -0.0001360439392286139
-2.00000000004387 -6.58047838442144e-10
-2.0 0.0
```

Out[8]: (-2.0, 0.0)



In [ ]:
```
1
```

Quasi-Newton Method

The derivative $f'(x)$ may not be always available. In that case we can use the difference at two points to approximate the derivative.

In [9]:
```
def qnewton(f, x0, x1, *args, target=1e-6):
    """Quasi-Newton's method.
        f: no need to return df/dx
        x0, x1: initial guess at two points
        *args: parameter for f(x,*args)
        target: accuracy target"""
    y0 = f(x0, *args)
    y1 = f(x1, *args)
    x, y = x1, y1
    while abs(y) > target:
        dy = (y1 - y0)/(x1 - x0)  # approximate derivative
        x = x1 - y1/dy     # update x
        plt.plot([x0,x1,x], [y0,y1,0], '-+')
        y = f(x, *args)   # new y
        print(x, y)
        x0, y0, x1, y1 = x1, y1, x, y
    return(x, y)
```

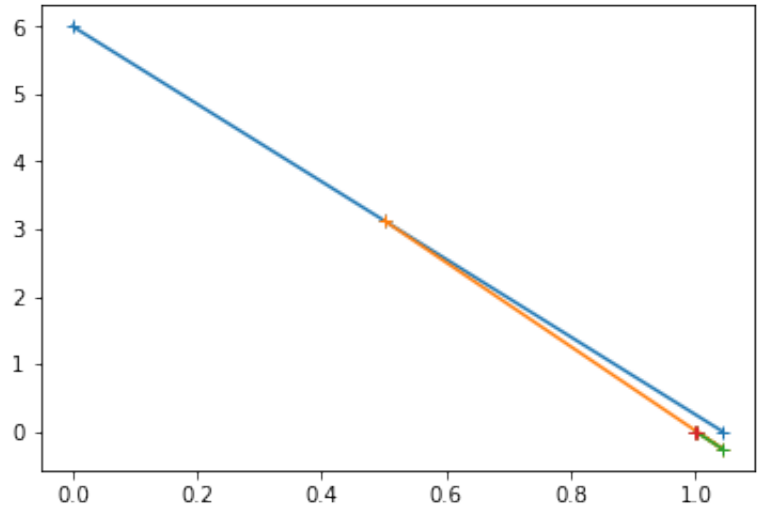
In [10]:
```
1 qnewton(poly, 5, 3, a, False) # no need of gradient
```

Out[10]: (3, 0)

In [11]:
```
1 qnewton(poly, 0, 0.5, a, False)
```

```
1.0434782608695652 -0.25889701652009545
1.0018975332068312 -0.011381591776396638
0.999985511487325 8.693128596459765e-05
1.0000000045921447 -2.755286776512378e-08
```

Out[11]: (1.0000000045921447, -2.755286776512378e-08)



Define your own function and see how Newton's or Quasi-Newton method works.

In [ ]:
```
1
```

Discrete-time Dynamics

We have already seen the iterative dynamics by multiplication of a matrix, which can cause expansion, shrinkage, and rotation.
With nonlinear mapping, more varieties of behaviors including chaos can be observed.

1D Dynamics

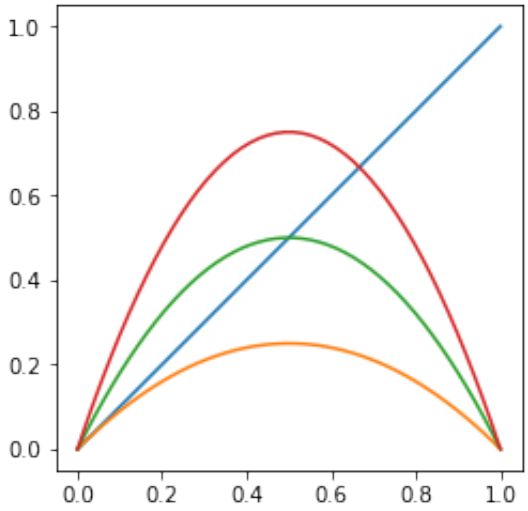
The simplest case is the logistic map.

$$x_{k+1} = ax_k(1 - x_k)$$

In [12]:
```
1 def logistic(x, a):
2     return(a*x*(1 - x))
```

In [13]:
```
x = np.linspace(0, 1, 50)
y1 = logistic(x, 1)
y2 = logistic(x, 2)
y3 = logistic(x, 3)
plt.plot([0,1], [0,1])
plt.plot(x, y1, x, y2, x, y3)
plt.axis('square')
```

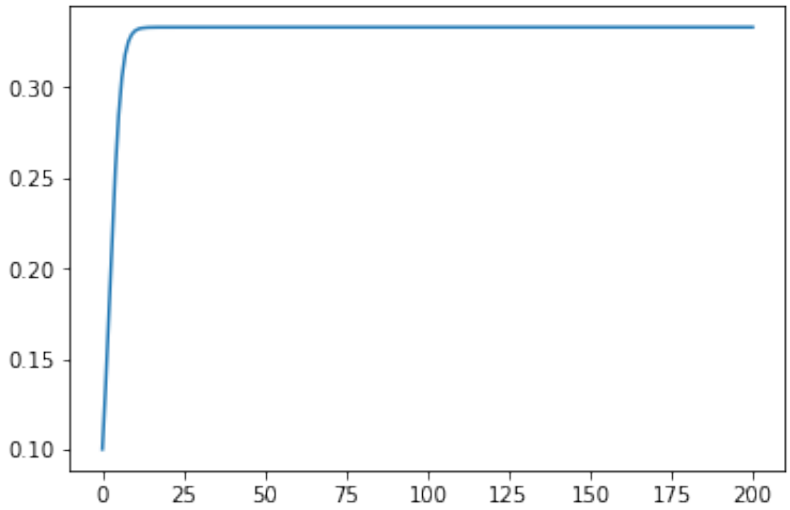
Out[13]: (-0.05, 1.05, -0.05, 1.05)



In [14]:
```
def iterate(f, x0, a, steps=100):
    """x0: initial value
        a: parameter to f(x,a)"""
    x = np.zeros(steps+1)
    x[0] = x0
    for k in range(steps):
        x[k+1] = f(x[k], a)
    return(x)
```

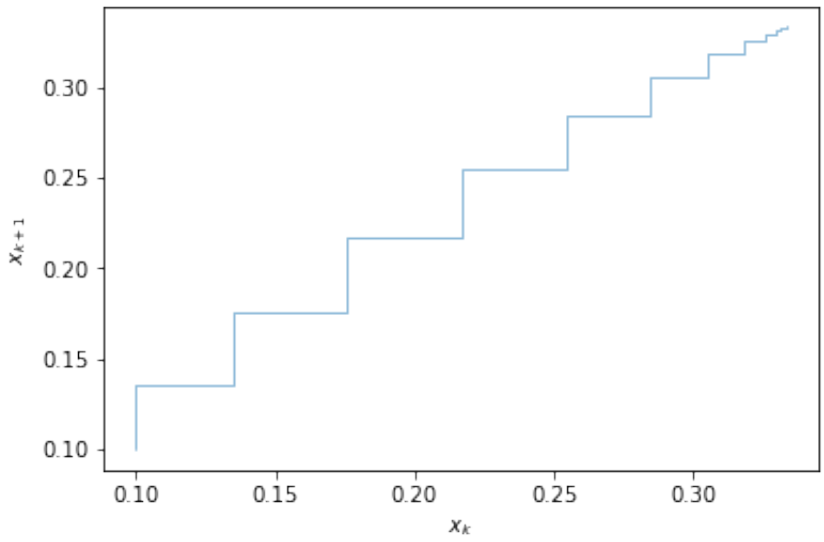
Try iteration with different parameter a.

In [15]:
```
1 x = iterate(logistic, 0.1, 1.5, 200)
2 plt.plot(x);
```



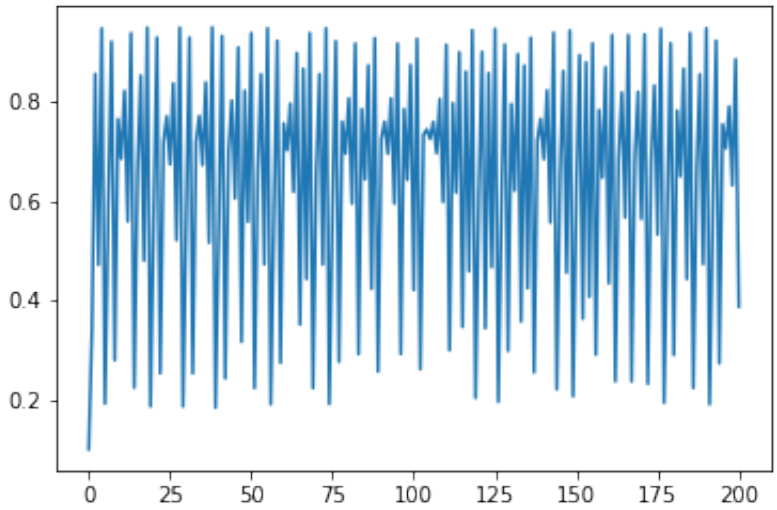
Trajectory in x_k-x_{k+1} plane, called "cobsplot".

In [16]:
```
1 x2 = np.repeat(x, 2)  # duplicate items
2 plt.plot(x2[:-1], x2[1:], lw=0.5) # (x0,x1), (x1,x1), (x1,x2),...
3 plt.xlabel('$x_k$'); plt.ylabel('$x_{k+1}$');
```

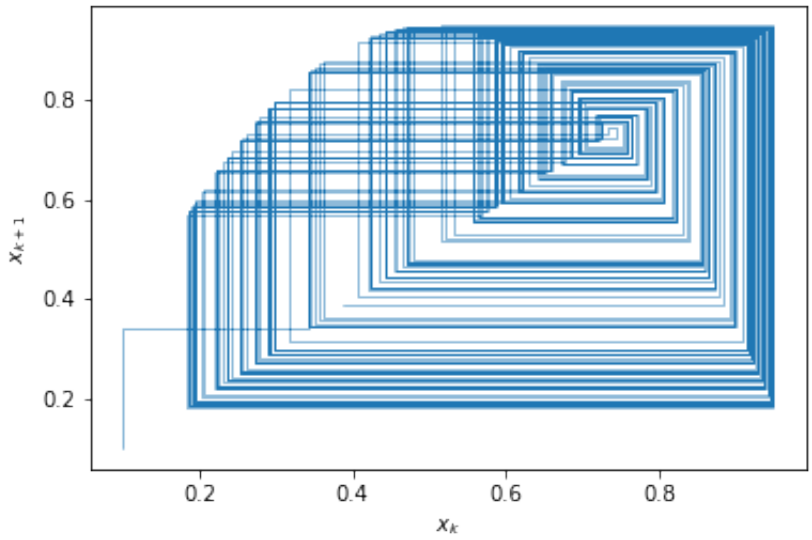


It is known that $3.57 < a < 4$ can cause chaotic oscillation.

In [17]:
```
1 x = iterate(logistic, 0.1, 3.8, 200)
2 plt.plot(x);
```



In [18]:
```
1 x2 = np.repeat(x, 2)
2 plt.plot(x2[:-1], x2[1:], lw=0.5)
3 plt.xlabel('$x_k$'); plt.ylabel('$x_{k+1}$');
```



In [ ]:
```
1
```

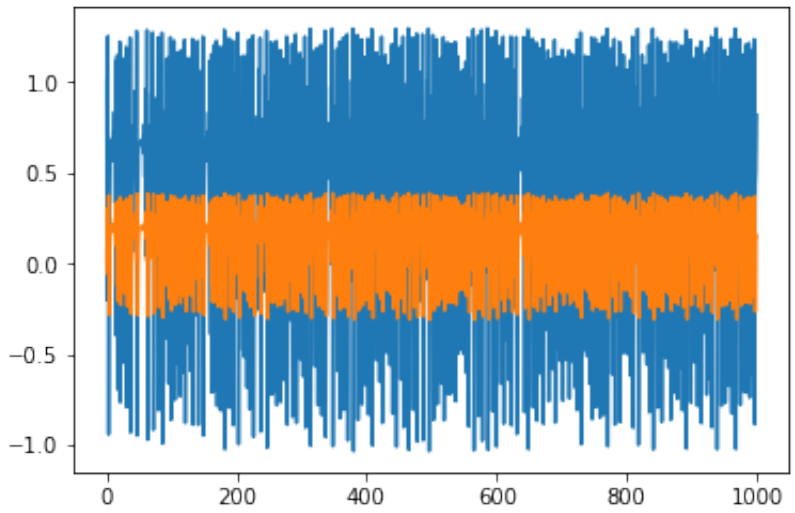
2D Dynamics

Let us see an example: Hénon map.

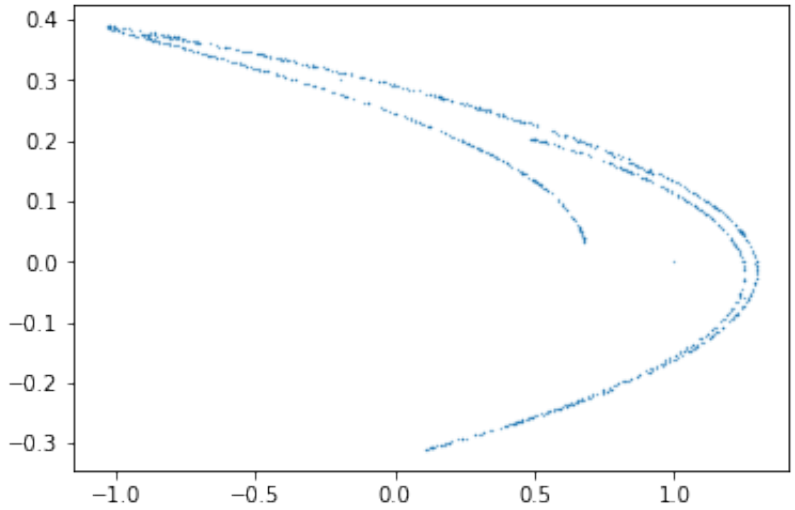
In [19]:
```
def henon(xy, ab=[1.4, 0.3]):
    """Henon map: stretch in y and fold to x
        x_{k+1} = 1 - a*x_k**2 + y_k
        y_{k+1} = b*x_k
        xy: state [x, y]
        ab: parameters [a, b]        xy: state [x, y]
    """
    x, y = np.array(xy, dtype=float)  # unpack array
    a, b = np.array(ab, dtype=float)
    x1 = 1 - a*x**2 + y
    y1 = b*x
    return(np.array([x1, y1]))
```

In [20]:
```
def iterate_vec(f, x0, a, steps=100):
    """x0: initial vector
        a: parameter to f(x,a)"""
    n = len(x0)  # state dimension
    x = np.zeros((steps+1, n))
    x[0] = x0
    for k in range(steps):
        x[k+1] = f(x[k], a)
    return(x)
```

In [21]:
```
x = iterate_vec(henon, [1, 0], [1.2, 0.3], 1000)
plt.plot(x);
```



In [22]:
```
1 plt.plot(x[:,0], x[:,1], '.', markersize=0.5);
```



In [ ]:
```
1
```

Iterated Function System

Iterated functions can also show fractal images. *Barnsley's fern* is made from a union of contracting maps.

$$x_{k+1} = \bigcup_i A_i x_k + b_i$$

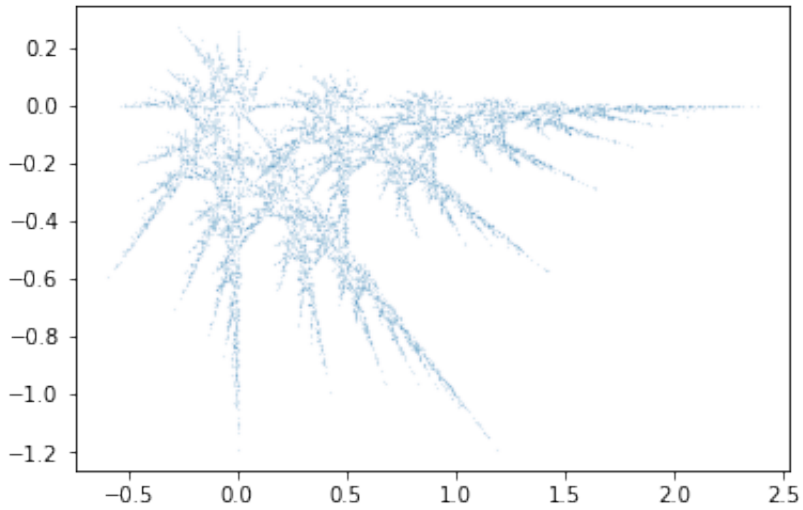
In [23]:
```
1 def barnsley(x, A):
2     """Barnley's fern:
3         x: initial point in 2D
4         a: matrices [A_0,...,A_m-1]"""
5     m = A.shape  # should be (m, 2, 3), last column for bias
6     i = np.random.randint(m[0])  # choose one
7     y = A[i] @ np.append(x, 1)
8     return(y)
```

In [24]:
```
1 # Example
2 A = np.array([
3     [[0.8, 0, 0.5],
4      [0, 0.5, 0]],  ## shift
5     [[0.5, 0.5, 0],
6      [-0.5, 0.5, 0]],  ## rotate
7 ])
```

In [25]:
```
1 barnsley([1, 0], A)
```

Out[25]:
```
array([1.3, 0. ])
```

In [26]:
```
x = iterate_vec(barnsley, [0, 0], A, 10000)
plt.plot(x[:,0], x[:,1], '.', markersize=0.1);
```



In [ ]:
```

```

Exercise

1. Newton's Method

Newton's method can be generalized for n dimensional vector $x \in \mathfrak{R}^n$ and n dimensional function $f(x) = \mathbf{0} \in \mathfrak{R}^n$ as

$$x_{k+1} = x_k - J(x_k)^{-1} f(x_k)$$

where $J(x)$ is the *Jacobian matrix*

$$J(x) = \begin{pmatrix} df_1/dx_1 & \cdots & df_1/dx_n \\ \vdots & & \vdots \\ df_n/dx_1 & \cdots & df_n/dx_n \end{pmatrix}$$

1) Define a function that computes

$$f(x) = \begin{pmatrix} x_1^2 + x_2^2 - 1 \\ a_1 x_1 + a_2 x_2 - 1 \end{pmatrix}$$

and its Jacobian.

In [ ]:
```
def f(x, a, deriv=True):

```

In [ ]:
```
f([1,1], [1, 2])
```

2) Implement Newton's method for vectors and test how it works, e.g. for $a = [1, 2]$ with different initial guesses.

In [ ]:
```
def newton(f, x, a, target=1e-6):
    """Newton's method.
        f: should also return Jacobian matrix
        x: initial guess
        a: parameter for f(x,a)
        target: accuracy target"""
```

In [ ]:
```
newton(f, [2, 0], [1, 2])
```

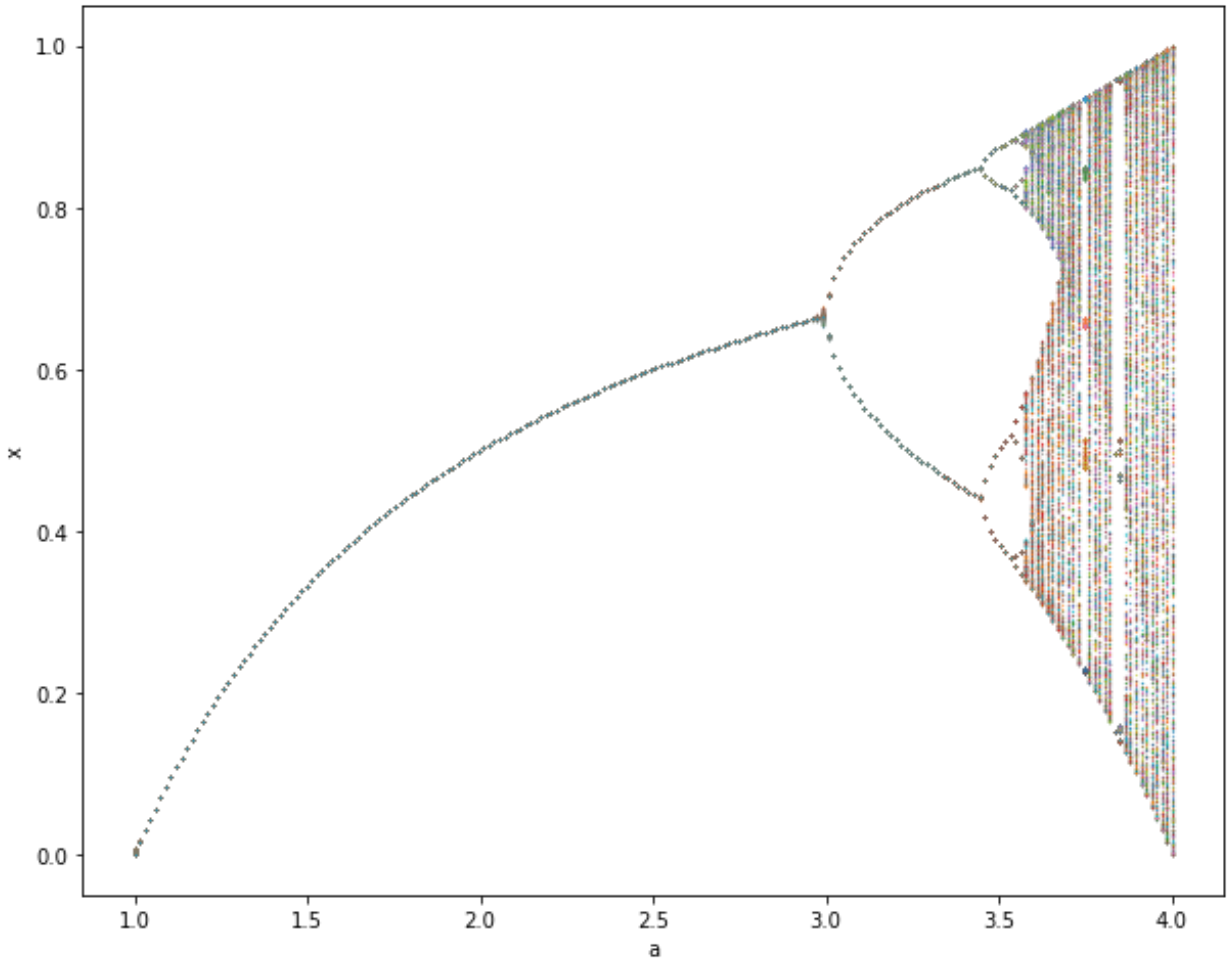
In [ ]:
```
newton(f, [0, 1], [1, 2])
```

2. Bifurcation and Chaos

A *bifurcation diagram* is a plot of trajectories versus a parameter.
1) For the logistic map

$$x_{k+1} = ax_k(1 - x_k)$$

draw the bifurcation diagram for parameter
a
$(1 \leq a \leq 4)$, like below:



Hint:

- Use the `logistic()` and `iterate()` functions from the previous lecture.
- For each value of a, show the trajectory (i.e., the values that x_k took over some iterations) of the map after an initial transient. Since x_k is 1D, you can plot the trajectory on the y axis. For example, take 200 points in $1 \leq a \leq 4$, run 1000 step iterations for each a, and plot x after skipping first 100 steps.

In [ ]:
```
1 def logistic(x, a):
```

In [ ]:
```
def iterate(f, x0, a, steps=100):
    """x0: initial value
        a: parameter to f(x,a)"""
```

In [ ]:
```
n = # points in a
a =
s = # steps for each a
x =
```

2) Explain, in intuitive terms:

- How does the logistic map behave when $1 < a < 3$? Does anything change when you change the initial value of x_k?
- How does the logistic map behave when $3.57 \leq a \leq 4$?
- What is happening around $3.4 \leq a \leq 3.57$? Create more zoomed-in plots, sampling more values of a from this region, and describe the trajectories.

In [ ]:
```

```

3) A value of x_k that stays unchanged after applying a map f to it (i.e. $x_k = f(x_k) = x_{k+1}$) is called a "fixed point" of f.

Plot $x_{k+1} = ax_k(1 - x_k)$ along with $x_{k+1} = x_k$ for $a = 0.5, 2, 3.3$. What are the fixed points of these maps?

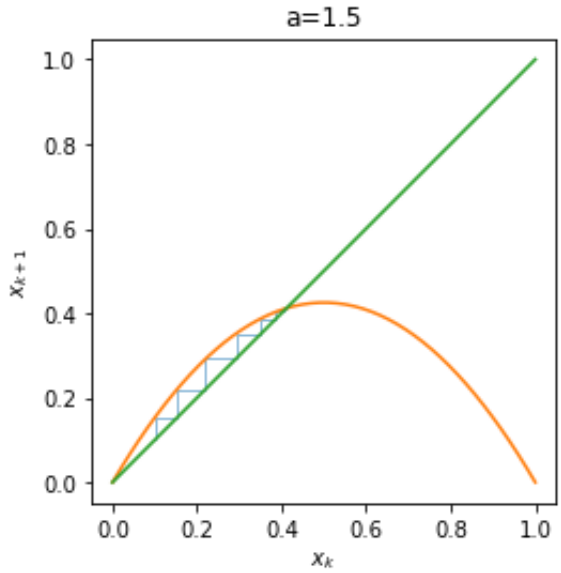
In [ ]:
```

```

4) A fixed point is said to be "stable" when nearby values of x_k also converge to the fixed point after applying f many times; it's said to be "unstable" when nearby values of x_k diverge from it.

Draw "cobweb plots" on top of each of the three plots in Q2-2 to visualize trajectories, as shown here for $a = 1.5$ (Use the code provided in the previous lecture notes; zoom in as needed).

Try several different initial values of x_k. Are the fixed points you found stable or unstable? Compute the slope (derivative) of $f(x_k) = ax_k(1 - x_k)$ at the fixed points.

How is stability of a fixed point
x^* related to
$f'(x^*)$?
What changes at
$a = 1$ and
$a = 3$?



In [ ]:
```
1
```

3. Iterated Function System

Design your own matrices for IFS to create a beautiful fractal figure, e.g., leaves, trees, neurons, etc.

Hint:

- use the iterate_vec() and barnsley() functions from the previous lecture.
- start by modifying matrix A from the example in the previous lecture.

In [ ]:
```
def iterate_vec(f, x0, a, steps=100):
    """x0: initial vector
        a: parameter to f(x,a)"""
```

In [ ]:
```
def barnsley(x, A):
    """Barnley's fern:
        x: initial point in 2D
        a: matrices [A_0,...,A_m-1]"""
```

In [ ]:
```
A =
```

In [ ]:
```
x = iterate_vec(barnsley, [0, 0], A, 10000)
plt.plot(x[:,0], x[:,1], '.', markersize=0.5)
```

In [ ]:
```

```